SPARQL-DL Queries for Antipattern Detection

Catherine Roussey[1], Oscar Corcho[2], Ondřej Šváb-Zamazal[3], François Scharffe[4], and Stephan Bernard[1]

[1] Irstea, 24 Av. des Landais, BP 50085, 63172 Aubiére, France
catherine.roussey@irstea.fr
[2] Ontology Engineering Group, Departamento de Inteligencia Artificial, Universidad Politécnica de Madrid, Spain
ocorcho@fi.upm.es
[3] Knowledge Engineering Group, University of Economics Prague, Czech Republic
ondrej.zamazal@vse.cz
[4] LIRMM, Université de Montpellier, France
scharffe@lirmm.fr

Abstract. Ontology antipatterns are structures that reflect ontology modelling problems, they lead to inconsistencies, bad reasoning performance or bad formalisation of domain knowledge. Antipatterns normally appear in ontologies developed by those who are not experts in ontology engineering. Based on our experience in ontology design, we have created a catalogue of such antipatterns in the past, and in this paper we describe how we can use SPARQL-DL to detect them. We conduct some experiments to detect them in a large OWL ontology corpus obtained from the Watson ontology search portal. Our results show that each antipattern needs a specialised detection method.

Keywords: OWL, ontology, antipattern, SPARQL, SPARQL-DL

1 Introduction

In knowledge engineering, the concept of knowledge modelling pattern or ontology design pattern is used to refer to modelling solutions that allow solving recurrent knowledge modelling or ontology design problems, as defined in [14]. Antipatterns are defined as patterns that appear obvious but are ineffective or far from optimal in practice, representing worst practices about how to structure and design an ontology. However, in contrast to ontology design patterns, which are rooted deeply in the most recent ontology engineering methodologies, the work on antipatterns is less detailed. Antipatterns may have several applications: Antipatterns can be used to train and guide new ontology developers. Ontology editors can incorporate antipattern detection tool in order to detect potential errors during ontology development. Most of ontology systems that deal with a large set of ontologies, like ontology retrieval systems, need ontology quality measures. The quality of ontology can be evaluated by computing the number of antipattern occurrences. As far as we know antipattern studies are mainly applied to ontology debugging tasks. One of the earliest works in

this direction was set by the OntoClean method [10], which defined a set of meta-properties applied to classes and a set of procedures to check and correct the subsumption relations between classes. Other sources for antipatterns are: [19], which proposes four terminological patterns applied on class names to detect possible errors in subsumption relations between classes. The Laboratory of Applied Ontology has identified four logical antipatterns called MixedDomains, all of them focused on property domains and ranges. And [15] describes common difficulties for newcomers to Description Logics (DL) in understanding the logical meaning of expressions.

Several tools can be used for antipattern detection, most of which are available inside ontology editors and require the use of a reasoner to provide their justifications. Pellint[8] focuses on the detection and repair of antipatterns to improve ontology reasoning performance. The Protégé Explanation Workbench [11] and SWOOP [13] provide justifications of inconsistencies in ontologies based on the outputs of DL reasoners. However, using a reasoner for this purpose is not always possible, since in some complex ontologies, where the number of errors is too high, reasoners fail to provide any results. Besides, the catalogue of antipatterns or errors that they can detect is fixed.

Our antipattern detection methods follow a more general approach. They can work with an extensible set of antipatterns and some of them can be applied without the use of a reasoner. In general, our approach is based on the use of a set of SPARQL-DL queries for each antipattern to be detected. Then, each SPARQL-DL query is translated into SPARQL one. In our process, we can decide whether inferences are enabled or not before running any SPARQL queries, and we also offer the possibility of transforming the original ontologies into a form where SPARQL queries should retrieve more results.

We first tested our methods on the detection of one complex antipattern using only a set of SPARQL queries [16]. This first experiment was applied on 5 ontologies. This paper presents a larger experiments using a more generic approach: More antipatterns are detected on a larger set of ontologies. We also try to simplify the creation of queries using SPARQL-DL language. One of our final goal is to understand how often antipatterns appear in existing publicly available ontologies.

This paper is structured as follows. Section 2 briefly describes the antipatterns that will be used to run our experiments. Section 3 will describe the methods we have followed in order to run the experiments. Section 4 describes the experiment setup and the results of the experimentation. Finally, Section 5 provides some conclusions to the work done, based on the experiment results, and outlines the next steps to be done in our work.

2 A catalogue of antipatterns

A set of patterns commonly used by domain experts in their implementation of OWL ontologies are identified in [9]. These patterns resulted in unsatisfiable classes or modelling errors, due to misuse or misunderstanding of DL expressions.

In this section we will describe 4 antipatterns which are the ones that, as our experience has shown, are easier to understand and debug by domain experts. These patterns are categorized into two groups by [9]:

- *Detectable Logical AntiPatterns (DLAP)*: this type of patterns generates unsatisfiable classes that are normally identified by existing ontology debugging tools, although the information provided back to the user is not described according to such a pattern. This makes it sometimes difficult for ontology developers to find a good solution [11], [13].
- *Cognitive Logical AntiPatterns (CLAP)*: they represent possible modelling errors that may be due to a misunderstanding of the logical consequences of the used expression. This type of patterns is not detected by debugging tools, although in some cases their combination may lead into unsatisfiable classes that are detected.

Now we briefly[5] describe them from the simplest to the most complicated one. Each description contains:

- name, acronym and category that they belong to (i.e. DLAP or CLAP),
- several formal descriptions using the german syntax of DL [6],
- brief explanation of why this antipattern can appear.

SynonymOrEquivalence (SOE) antipattern – CLAP category

$$C_1 \equiv C_2; \tag{1}$$

The ontology developer wants to express that two classes C_1 and C_2 are identical, something which is not particulary useful especially if the ontology does not import others. Indeed, what the ontology developer generally wants to represent is a terminological synonymy relation, i.e. a class has two labels: the labels associated (or used as) the name of the classes C_1 and C_2. Usually one of these classes is not used anywhere else in the axioms defined in the ontology.

EquivalenceIsDifference (EID) antipattern – DLAP category

$$C_1 \equiv C_2;\ \ Disj(C_1, C_2); \tag{2}$$

$$C_1 \sqsubseteq C_2;\ \ Disj(C_1, C_2); \tag{3}$$

where the notation $Disj(C_1, C_2)$ is as a shorthand for $C_1 \sqcap C_2 \sqsubseteq \bot$.

From our experience in ontology debugging, we notice that this antipattern comes from a misunderstanding of the subClassOf relation. When the ontology developer has explicitly expressed that C_1 and C_2 are equivalent and disjoint, (s)he wants to say that C_1 and C_2 share some common properties and C_1 has more properties than C_2 (or vice versa). After a short training session the developer would discover that (s)he really wants to express that C_1 is a subclass of C_2 ($C_1 \sqsubseteq C_2$). Another possibility is that the ontology developer explicitly expressed that C_2 is a parent class of C_1. But, these two classes are determined as disjoint from each other by a reasoner.

[5] Additional information, such as examples and SPARQL queries, are available at [7].
[6] antipatterns are abstract structures that can have several DL forms.

AndIsOr (AIO) antipattern – DLAP category

$$C_3 \sqsubseteq C_1 \sqcap C_2;\ \ Disj(C_1, C_2); \tag{4}$$
$$C_3 \equiv C_1 \sqcap C_2;\ \ Disj(C_1, C_2); \tag{5}$$
$$C_3 \sqsubseteq \exists R.(C_1 \sqcap C_2);\ \ Disj(C_1, C_2); \tag{6}$$
$$C_3 \equiv \exists R.(C_1 \sqcap C_2);\ \ Disj(C_1, C_2); \tag{7}$$

This antipattern appears due to the fact that in common linguistic usage, "and" and "or" do not correspond consistently to logical conjunction and disjunction respectively [15]. An example is presented in [7].

OnlynessIsLoneliness (OIL) antipattern – DLAP category

$$C_3 \sqsubseteq \forall R.C_1; C_3 \sqsubseteq \forall R.C_2;\ \ Disj(C_1, C_2); \tag{8}$$
$$C_3 \equiv \forall R.C_1; C_3 \sqsubseteq \forall R.C_2;\ \ Disj(C_1, C_2); \tag{9}$$
$$C_3 \equiv \forall R.C_1; C_3 \equiv \forall R.C_2;\ \ Disj(C_1, C_2); \tag{10}$$

C_1 and C_2 are defined as disjoint. The ontology developer created an universal restriction to say that instances of C_3 can only be linked with property R[7] to instances of C_1. Next, a new universal restriction is added saying that instances of C_3 can only be linked with R to instances of C_2. During a long development process, the ontology developer forgot the previous axiom that can be inherited from any of the parent classes.

3 SPARQL-based Detection of Ontology Antipatterns

In this section we describe the different methods that we have elaborated in order to detect antipatterns in OWL ontologies by means of SPARQL and SPARQL-DL queries, based on the usage of the PatOMat ontology pattern detection tool [3]. This tools is part of the PatOMat suite of tools, which is focused on the detection of patterns in ontologies and their transformation. This detection tool is based on Jena 2.6.2[1] and Pellet 2.0.1[5], and enables the processing of a set of SPARQL queries over a set of ontologies, producing a report in terms of numbers of patterns detected and details for each ontology. It processes either only asserted axioms or both inferred and asserted axioms of given ontology.

Using this tool we are querying an OWL ontology by means of a query language (SPARQL) that is agnostic about the underlying knowledge representation model of OWL, i.e. we are actually querying the RDF serialization of an OWL ontology. There are also other available options in the current state of the art for OWL ontology pattern detection and transformation. First, there is OPPL language and its associated tools described by [12]. This language enables axiom-based manipulation with an OWL ontology. Second, there is a

[7] To be detectable, property R must have at least a value, normally specified as a (minimum) cardinality restriction for that class, or with existential restrictions.

language alternative for an OWL ontology querying Terp [4] which is based on the OWL Manchester syntax. While SPARQL is the language dedicated to query RDF triples, OPPL and Terp are dedicated to query the RDF serialization of OWL expressions because they contain OWL constructs like subClassOf, ComplementOf, DisjointWith. Nevertheless, in order to make queries easier (using some shortcuts for DL expressions in RDF syntax, e.g. omitting RDF collection vocabulary) we used SPARQL-DL abstract syntax defined in [17]. This enables us to express queries in more compact way. To plug in such queries into our approach we developed a query translator that transforms an input query in SPARQL-DL abstract syntax into a SPARQL query. SPARQL-DL queries and SPARQL queries are available on our antipattern web-page [7].

Transforming antipatterns into SPARQL-DL queries is not a trivial task. For each antipattern, several SPARQL-DL queries are needed to detect antipattern occurrences in OWL class definition. The difficulties come from several points:

- An antipattern can be associated to several logical formulae in DL syntax. For example, we presented 3 formulae for OIL antipatterns.
- Some logical formulae are composed of several atomic axioms. We defined an atomic axiom as a condition (necessary $\sqsubseteq$ or sufficient $\equiv$) associated to a named class C using at most one constructor ($\forall$, $\exists$, $\neg$ or $\sqcap$) and its associated operands: one class and one property for $\forall$, $\exists$ and two classes for $\sqcap$. All these classes should be named classes. An example of atomic axiom can be $C \sqsubseteq \exists R.X$. For example, the 3 formulae of the OIL antipattern contain 3 atomic axioms.
- Ontology developers can have very different implementation styles when designing an OWL ontology. For example, some developers prefer to write long class definitions. In that case, a class is defined by a conjunction of unamed classes: $C \sqsubseteq (\exists R.X) \sqcap (\forall R.Y) \sqcap \ldots$. Others can prefer to write short definitions. A class is defined by a set of atomic axioms: $C \sqsubseteq \exists R.X; C \sqsubseteq \forall R.Y; C \sqsubseteq \ldots$. Thus, in the case of an antipattern formula, an atomic axiom can be located at different places in the class definition.
- An atomic axiom can belong to the class definition or can be inherited from a parent class definition.
- An atomic axiom can be stated by the ontology developer or inferred by a reasoner.

To build our queries, we first imagine different versions of each antipattern formulae using the SPARQL-DL abstract syntax. We try to imagine where an atomic axiom can be stated by the ontology developer in a class definition. We limit our imagination to class definitions that have at most four conjunctions. We also try to imagine the different manner to express a disjoint axiom. We take in account the fact that:

- disjoint axioms are symetric $Disj(C_1, C_2) \vDash Disj(C_2, C_1)$,
- disjoint axiom can be inferred from a logical negation $C_1 \sqsubseteq \neg C_2 \vDash Disj(C_1, C_2)$.

Then we automatically translate each SPARQL-DL queries into SPARQL ones. Notice that a SPARQL-DL query is just a simplification of a SPARQL

query, which represent an exact translation of the previous one. We also automatically generate SPARQL queries which merges all the different versions.

Now we will describe four methods that we have followed in order to detect antipatterns in the ontology corpus. Overall workflow of our approach is depicted in Figure 1.

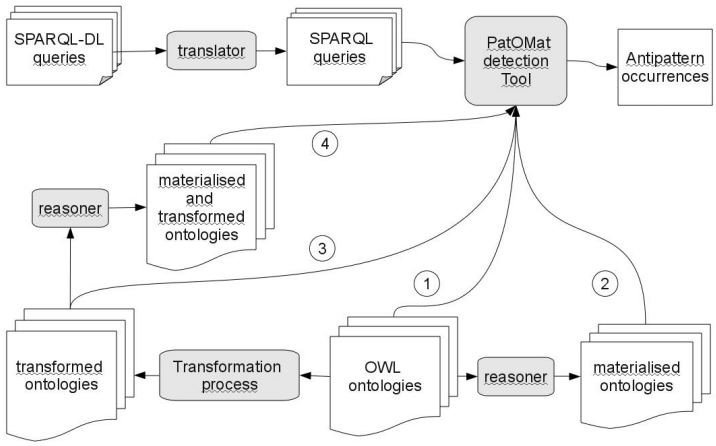

Fig. 1. The antipattern detection methods

Method 1: Use of SPARQL Queries over Asserted OWL Ontology Axioms In this approach, we take into account that SPARQL query engines per se do not consider inferences that can be done with OWL ontologies. However, our assumption is that there will be cases where ontologies cannot be processed by a reasoner or the reasoner results cannot be obtained in a reasonable time. This normally happens with large ontologies or with ontologies with a large number of errors. For example when several transitive properties are used in numerous class definitions, the reasoner reaches an out of memory alarm.

Method 2: Use of SPARQL Queries over Inferred and Asserted Ontology Axioms If it is possible to use a reasoner, we materialise all the inferences that can be done by an OWL reasoner on the ontologies and then run SPARQL queries over the resulting ontologies, called materialised ontologies.

Method 3 and 4: Use of SPARQL Queries over Transformed OWL Ontologies Due to the complexity of creating a large number of SPARQL-DL queries for an antipattern and to the fact that different ontology developers may have different

implementation styles, we propose to follow a two steps process where we apply transformations before executing the queries. Transformations have two goals: to harmonise the implementation style of the ontology and to simulate some of the axioms inferred by a reasoner. This last goal is useful only for ontologies that can not be processed by a reasoner.

The current transformations that we apply are:

- If the ontology contains $C_1 \equiv C_2$ where C_1 and C_2 are named classes, we add two new axioms $C_1 \sqsubseteq C_2$ and $C_2 \sqsubseteq C_1$.
- If a named class is defined by conjunction of named or unnamed classes, we split it into several simpler axioms. E.g., considering the following class definition: $C \sqsubseteq X \sqcap Y$, in that case we add two axioms $C \sqsubseteq X$ and $C \sqsubseteq Y$.
- If a parent class contains an axiom, we add it also in its direct child class. E.g., considering the following definition of the class: $C_1 \sqsubseteq \exists R.X$. If $C_{1.1}$ is a direct child of C_1, $C_{1.1} \sqsubseteq C_1$, we add the axiom $C_{1.1} \sqsubseteq \exists R.X$. In this case, the transformation is not repeated over the class hierarchy.

In this paper, we have explored the behaviour of the SPARQL query detection method both after transformation on the asserted ontology and on the materialised ontology.

4 Experimentation: Finding Antipatterns in Real-world Ontologies

In this section, we describe the results of our experiments with a corpus of ontologies downloaded directly from the Web and by the Watson semantic search engine. We will first describe how we have built the ontology corpus, and then we present the results of applying the different methods from Section 3 over this ontology corpus.

4.1 Building a Corpus of (Debuggable) OWL Ontologies

We have used the Watson API [6] to retrieve publicly available ontologies and we have always accessed these ontologies using the Watson cache, since there are sometimes mismatches between the stored URIs of those ontologies and the actual files that can be obtained. We searched ontologies satisfying the following two constraints: they should be represented in OWL and they should have at least five classes. We collected 2927 unique ontologies. Next, we checked the consistency of all these ontologies using the Pellet reasoner, and 71 of them were classified as inconsistent.[8] From inconsistent ontologies, we removed the whole ABox so that it is possible to use a reasoner as proposed in our second method (none of our antipatterns considers the ABox, and hence the removal of the ABox

[8] We use the definition of inconsistency proposed by [18]. An ontology is inconsistent, if there is no interpretation that is a model for it. An ontology is incoherent if it contains at least one unsatisfiable class.

does not have any impact on the results obtained). This was done by OWL-API [2], which resulted in five less ontologies, since they were not parsable by this API. Consequently, the corpus is composed of 66 incoherent ontologies, that is, 66 ontologies that contain at least one unsatisfiable class.

From these ontologies we built three sets of ontologies [9]:

1. *Antipattern ontologies*: 5 ontologies in this set have already been used for the creation and update of the antipattern catalogue presented in [9]. It contains the HydrOntology (which has 159 classes whose 114 are unsatisfiables), the Forestal Ontology (which has 93 classes whose 62 are unsatisfiable), the Tambis ontology (which has 395 classes whose 112 are unsatisfiable), the Sweet Numeric ontology (which has 2364 classes whose 2 are unsatisfiable) and the University ontology of the MIND Lab (which has 29 classes whose 7 are unsatisfiable). Notice that in our experiment Hydrontology and the Tambis ontologies cannot be processed by Pellet in a reasonable time.
2. *W3C/DL ontologies*: we noticed that 31 ontologies were build by DL experts in order to test reasoner performance and results. These ontologies are characterized by having less than 18 classes (whose at most 4 classes are unsatisfiables ones). The axioms contained are very complex: inverse properties, functional properties, lots of conjunctions or disjunctions etc. But all these ontologies can be processed by Pellet.
3. *Web ontologies*: this set contains heterogeneous ontologies from various domains. There are huge ontologies which contain more than 1000 classes and Pellet cannot process them in a reasonable time, e.g. an old version of the Open CyC ontology, the Computer Science for Non-Computer Scientists ontology. There are also medium size ontologies where the number of classes is up to 100 which Pellet can process, e.g. Ontubi (an Ontology for Ubiquitous Computing) or the wine ontology.

4.2 Experiments

We made the following experiments over the 3 sets of ontologies, using the antipattern detection methods described in Section 3:

1. *SP*: a detection in the original ontologies using SPARQL[10] queries and no inference (only with asserted axioms).
2. *SP+R*: a detection in the materialised ontologies (asserted and inferred axioms) using SPARQL queries after applying a reasoner (Pellet).
3. *SP_Trans*: a transformation of the original ontologies and detection using SPARQL queries and no inference (only with asserted axioms).
4. *SP_Trans+R*: a transformation of the original ontologies and detection in the materialisation of these harmonised ontologies after applying a reasoner.

[9] All of these ontologies are available from [7].

[10] Let us note that all SPARQL queries in our experiments were automatically converted from original SPARQL-DL queries.

In some of these experiments we also use the keyword MANUAL to refer to the manual detection process using the basic debugging tools provided by ontology editors. The manual detection method is applicable only on the antipattern and the W3C/DL ontologies sets. This detection method is a baseline with respect to what can be detected using current state of the art debugging tools.

Evaluation of Antipattern Detection Precision We have evaluated the precision of the antipattern detection process. We have analysed manually each of the ontologies in our three sets and have assigned to each set one of the following three values:

- *TI* (True Inconsistency): the antipattern occurrence participates in the unsatisfiability of classes or the modelling error.
- *UI* (Unknown Inconsistency): the antipattern occurrence may be linked to the unsatisfiability of classes or modelling error, but the evaluator is not sure about it. Notice that the evaluator may find difficult to make a choice when the debuggable version of the considered ontology is not available.
- *FI* (False Inconsistency): the antipattern occurrence does not participate in the unsatisfiability of classes or modelling error.

4.3 Results

SOE detection In this case we look for a single atomic axiom written by the ontology developer. Thus only one SPARQL query is necessary to retrieve SOE occurrences and only the SP experiment was made over all 3 sets of ontologies.

set	number (nr.) of results	nr. of TI	nr. of UI	nr. of FI	nr. of onto
antipattern	16	15	0	1	2
W3C/DL	1	0	1	0	1
web	12	10	2	0	2

Table 1. SOE antipattern detection.

Due to the simplicity of the SOE antipattern, the most suitable detection method is the first method. Neither reasoner nor transformation process is needed for the detection of the SOE antipattern. The SPARQL query associated to the SOE antipattern reached 86% of precision: 25 occurrences over 29 are classified as true inconsistencies.

EID detection The EID antipattern is composed of two atomic axioms, and two formulae are possible. We defined 8 SPARQL queries associated to this antipattern. We also use 4 detection methods. Our results are limited to the fact that some ontologies cannot be processed by a reasoner in a reasonable time.

set	method	nr. of results	nr. of TI	nr. of UI	nr. of FI	nr. of onto
antipattern	manual	14	-	-	-	4
antipattern	SP	7	7	0	0	2
antipattern	SP+R	5885	14	5871	0	2
antipattern	SP_Trans	7	7	0	0	2
antipattern	SP_Trans+R	5885	14	5871	0	2
W3C/DL	manual	8	-	-	-	4
W3C/DL	SP	4	4	0	0	4
W3C/DL	SP+R	48	7	41	0	4
W3C/DL	SP_Trans	5	5	0	0	4
W3C/DL	SP_Trans+R	48	7	41	0	4
web	SP	9	9	0	0	1
web	SP+R	126	0	126	0	1
web	SP_Trans	9	9	0	0	1
web	SP_Trans+R	132	0	132	0	1

Table 2. EID antipattern detection.

Table 2 shows the results of our detection methods. We notice that using a reasoner creates some unexpected occurrences of EID antipattern. Reasoner infers that an unsatisfiable class is an equivalent to another unsatisfiable class and they are also disjoint from each others. Thus using a reasoner is not a good solution for a detection of this antipattern. The transformation improves a little bit the detection of this antipattern. It seems to be a promising direction of future work that we should improve the transformation procedure to detect more EID occurrences.

AIO detection The AIO pattern is composed of 2 atomic axioms. In Section 2 we have presented 4 formulae but in theory more formulae are possible. We imagine that a class definition can be composed of maximum 4 conjunctions: $C_3 \sqsubseteq C_w \sqcap C_x \sqcap C_y \sqcap C_z$. We defined 24 SPARQL queries corresponding to the C_1 and C_2 classes at different location of formulae. The transformation process modified the AIO pattern. Thus we added new SPARQL queries associated to the new pattern: $C_3 \sqsubseteq C_1 \sqcap C_2 \vDash C_3 \sqsubseteq C_1; C_3 \sqsubseteq C_2$.

Table 3 shows the results of our detection methods for the AIO antipattern. In this case our results are far from optimal. None of our detection method can detect the AIO occurrences in the antipattern set. This is due to the incapacity of our method to detect the atomic axiom $Disj(C_1, C_2)$ without a reasoner. Notice that the second detection method detects all the occurrences of the AIO pattern on the W3C/DL set. Thus it means that this antipattern needs a reasoner to be accurately detected.

OIL detection The OIL pattern is composed of 3 atomic axioms. We have presented 3 formulae but more formulae are possible depending on the implementation style of an ontology developer [7]. For these formulae, we imagine that a class definition can be composed of two conjunctions parts. In all, we defined 84 SPARQL queries.

set	method	nr. of results	nr. of TI	nr. of UI	nr. of FI	nr. of onto
antipattern	manual	6	-	-	-	3
antipattern	SP	1	1	0	0	1
antipattern	SP+R	3	2	1	0	2
antipattern	SP_Trans	1	1	0	0	1
antipattern	SP_Trans+R	53761	0	53761	0	2
W3C/DL	manual	9	-	-	-	8
W3C/DL	SP	2	2	0	0	2
W3C/DL	SP+R	9	9	0	0	8
W3C/DL	SP_Trans	4	2	2	0	3
W3C/DL	SP_Trans+R	236	37	199	0	8
web	SP	0	0	0	0	0
web	SP_Trans	67	0	67	0	1

Table 3. AIO antipattern detection.

set	method	nr. of results	nr. of TI	nr. of UI	nr. of FI	nr. of onto
antipattern	manual	8	-	-	-	3
antipattern	SP	2	2	0	0	2
antipattern	SP+R	2	2	1	0	2
antipattern	SP_Trans	2	2	0	0	2
antipattern	SP_Trans+R	72	6	66	0	2
web and W3C/DL		0	0	0	0	0

Table 4. OIL antipattern detection.

Results from Table 4 are surprising. In the case of the antipattern ontologies set we notice that the disjoint atomic axiom was not detected because it is not stated by the ontology developer. Furthermore using a reasoner produces unexpected antipattern occurrences. Thus any of our detection methods is good enough to detect OIL antipattern. Maybe for this specific pattern it is not necessary to detect exactly all the atomic axioms of the pattern. We should limit our detection method to the beginning of the OIL pattern without the disjoint axiom.

5 Conclusion and Future Work

In this paper we have shown how antipatterns can be detected using different methods that are based on the use of SPARQL queries, OWL reasoners and transformation tools. First, we have presented several antipatterns. Second, we have proposed different detection methods. Then, we applied them on a set of publicly available inconsistent ontologies. Finally, we have tried to figure out what are the best detection methods to be used for each antipattern. In many cases these antipattern detection methods are very sensitive to the *implemen-*

tation style of the ontology developer. Thus we recommand to avoid long class definition and to limit the use of unamed classes. Our future work will focus on the refinement of the methods that we have proposed in this paper. We will also try to design new antipatterns and detect them appropriately.

Ondřej Šváb-Zamazal has been partially supported by CSF grant no. P202/10/1825.

References

1. Apache jena. `http://jena.apache.org/`, 2012.
2. The owl api. `http://owlapi.sourceforge.net/`, 2012.
3. Patomat ontology pattern detection tool. `http://owl.vse.cz:8080/DetectionTool/`, 2012.
4. Pellet 2.1: Introducing terp. `http://weblog.clarkparsia.com/2010/04/01/pellet-21-introducing-terp`, 2012.
5. Pellet: Owl 2 reasoner for java. `http://clarkparsia.com/pellet/`, 2012.
6. Watson: Exploring the semantic web. `http://watson.kmi.open.ac.uk/WS_and_API.html`, 2012.
7. web site related to our ontology antipattern detection methods. `https://sites.google.com/site/ontologyantipattern`, 2012.
8. K. Clark. Pellint: An ontology repair tool. `http://weblog.clarkparsia.com/2008/07/02/pellint-an-ontology-repair-tool/`, 2008.
9. O. Corcho, C. Roussey, L. M. Vilches Blázquez, and I. Pérez. Pattern-based OWL ontology debugging guidelines. In Proceedings of WOP, CEUR Workshop proceedings, pages 68–82, October 2009.
10. N. Guarino and C. A. Welty. Evaluating ontological decisions with OntoClean. Commun. ACM, 45(2):61–65, 2002.
11. M. Horridge, B. Parsia, and U. Sattler. Laconic and precise justifications in OWL. In Proceedings of ISWC, pages 323–338, 2008.
12. L. Iannone, A. L. Rector, and R. Stevens. Embedding knowledge patterns into OWL. In Proceedings of ESWC, pages 218–232, 2009.
13. A. Kalyanpur, B. Parsia, E. Sirin, and J. Hendler. Debugging unsatisfiable classes in OWL ontologies. Journal of Web Semantics, 3(4):268–293, 2005.
14. V. Presutti, A. Gangemi, S. David, G.A. de Cea, M.C. Suárez-Figueroa, E. Montiel-Ponsoda, and M. Poveda. NeOn deliverable D2. 5.1. a library of ontology design patterns: reusable solutions for collaborative design of networked ontologies. 2008.
15. A. L. Rector, N. Drummond, M. Horridge, J. Rogers, H. Knublauch, R. Stevens, H. Wang, and C. Wroe. OWL pizzas: Practical experience of teaching OWL-DL: common errors & common patterns. In Proceedings of EKAW, pages 63–81, 2004.
16. C. Roussey, O. Corcho, O. Šváb-Zamazal, F. Scharffe, and S. Bernard. Antipattern detection in web ontologies: an experiment using sparql queries. In proceedings of EGC, pages 321–326, 2012.
17. E. Sirin and B. Parsia. SPARQL-DL: SPARQL query for OWL-DL. In Proceedings of OWLED, 2007.
18. H. Stuckenschmidt. Debugging OWL ontologies - a reality check. In Proceedings of EON, 2008.
19. O. Šváb-Zamazal and V. Svátek. Analysing ontological structures through name pattern tracking. In Proceedings of EKAW, pages 213–228, 2008.